



A Python Primer for Mathematics

M. Messerschmidt

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



https://github.com/miekmesserschmidt/a_python_primer_for_math

A Python Primer for Mathematics

January 20, 2019

Contents

0.1 Introduction

This document is meant to prime mathematics students into using Python for doing mathematics symbolically and numerically. It is *not* meant to be used as a comprehensive text, but rather as a demonstrative cheatsheet to get up and running with the basics Python and for using Python for scientific computing as quickly as possible.

We will give very brief introductory demonstrations of the basics of the Python language, before moving on to demonstrations of some of the basic features of the packages *sympy*, *numpy*, and *matplotlib* which are increasingly used in modern scientific computing.

In []:

Chapter 1

Python language basics

1.1 Comments

In [1]: *# Everything on a line after a '#' is ignored by Python*

1.2 Basic calculations with numbers

Addition works as expected

In [1]: 1+2

Out[1]: 3

... so does subtraction

In [5]: 2.5 - 6

Out[5]: -3.5

... and division.

In [4]: 3/2

Out[4]: 1.5

The floor division operator `//` throws away everything after the decimal point (as in long division)

In [1]: 13 // 5 *# quotient of 13 divided by 5 (disregarding the fractional part or remainder)*

Out[1]: 2

... and the `%` operator gives the remainder of a division (as in long division)

In [6]: 13 % 5 *# remainder when 13 is divided by 5*

Out[6]: 3

The power operator `**` is often useful ($2^4 = 16$)

In [8]: 2**4

Out[8]: 16

1.3 Variables

We can assign values to variables

```
In [9]: a = 5
        b = 6
```

... and then compute with them:

```
In [11]: a + b
```

```
Out[11]: 11
```

We can reassign their values,

```
In [15]: a = 9
```

... to change the outcome of the computation

```
In [16]: a + b
```

```
Out[16]: 15
```

1.4 Strings (text)

Strings store text. We use either ' ... ' or " ... " to denote a string

```
In [17]: "This is a string"
```

```
Out[17]: 'This is a string'
```

```
In [18]: 'This is also a string'
```

```
Out[18]: 'This is also a string'
```

We can join strings using the "+" operator. This is called *concatenation*.

```
In [19]: "begin" + "ner" + "s"
```

```
Out[19]: 'beginners'
```

We can assign strings to variables...

```
In [20]: part1 = "This is a message "
        part2 = "for you"
```

... and concatenate the variables

```
In [22]: full_message = part1 + part2
        full_message
```

```
Out[22]: 'This is a message for you'
```

We can access individual characters of a string. (Remember that we index from starting from zero!)

```
In [23]: n = 2
        full_message[n]
```

```
Out[23]: 'i'
```

We can also multiply strings.
Plug your ears and go ...

```
In [24]: "la"*10
```

```
Out[24]: 'lalalalalalalalalala'
```

1.5 Unpacking

There is a more efficient way of writing

```
a = 1
b = 2
c = 3
d = 4
```

The following is called *unpacking*:

```
In [29]: a, b, c, d = 1,2,3,4
         # or
         a, b, c, d = (1,2,3,4)
```

```
In [30]: d
```

```
Out[30]: 4
```

We are not restricted to just numbers. We can unpack anything

```
In [31]: firstname, lastname, age = ("john", "von neumann", 103)
         lastname
```

```
Out[31]: 'von neumann'
```

```
In [32]: age
```

```
Out[32]: 103
```

1.6 The print function

We can output text to the screen using the "print" function

```
In [34]: print(5+4)
         print("This is a message")
         print("The answer is", 42)
```

```
9
```

```
This is a message
```

```
The answer is 42
```

1.7 The str, int and float functions

We can convert numbers to strings with the str function:

```
In [1]: str(12)
```

```
Out[1]: '12'
```

... and strings to numbers using the int or float functions:

```
In [2]: int("13")
```

```
Out[2]: 13
```

```
In [3]: float("1.111")
```

```
Out[3]: 1.111
```

1.8 Comparisons

We can ask Python if statements are true or false

```
In [35]: 4 < 6
```

```
Out[35]: True
```

```
In [36]: 4 <= 6
```

```
Out[36]: True
```

```
In [37]: 4 >= 6
```

```
Out[37]: False
```

```
In [38]: 5<5
```

```
Out[38]: False
```

```
In [39]: 5<=5
```

```
Out[39]: True
```

Notice the double equals "==" when asking if an equality is true:

```
In [40]: 3 == 3
```

```
Out[40]: True
```

```
In [41]: 3 == 4
```

```
Out[41]: False
```

... a single "=" will not work to compare numbers:

```
In [42]: 3 = 3
```

```
File "<ipython-input-42-49c8ce3fc03c>", line 2
3 = 3
    ^
```

```
SyntaxError: can't assign to literal
```

The "!=" operator means "not equal to"

```
In [43]: 3 != 4
```

```
Out[43]: True
```

We can also compare variables

```
In [44]: a,b,c = 5,6,7
```

```
In [45]: b != 7
```

```
Out[45]: True
```



```
In [46]: a < 5
```

```
Out[46]: False
```

```
In [47]: a <= 5
```

```
Out[47]: True
```

```
In [48]: b <= a
```

```
Out[48]: False
```

```
In [49]: a < b < c
```

```
Out[49]: True
```

We can also do computations in comparisons. Is the remainder when dividing by 2 equal to zero, i.e., Is b even? Is c even?

```
In [51]: b % 2 == 0      # 6 is even
```

```
Out[51]: True
```

```
In [53]: c % 2 == 0      # 7 is odd
```

```
Out[53]: False
```

Chapter 2

If statements

With if statements we can control the flow of execution of a program.

2.1 If ... statements

```
In [2]: a,b = 5,6
        if a == b:
            # this is not executed because 'a == b' is false
            print("a is equal to b")

        if a <= b:
            # this is executed because 'a <= b' is true
            print("a is less than or equal to b")
```

a is less than or equal to b

2.2 If ... else ... statements

```
In [2]: a,b = 5,6
        if a == b:
            print("a is equal to b")
        else:
            # this is only executed if a == b is false
            print("a is not equal to b")
```

a is not equal to b

2.3 If ... elif ... else ... statements

```
In [3]: name = "bobby"

        if name == "alice":
            print("Hi Alice")
        elif name == "bobby":
            print("Hi Bob")
        elif name == "richard":
```

```
    print("Hi Ricky")
else:
    print("Hi Stranger")
```

Hi Bob

Chapter 3

Lists

Lists are a fundamental data structure in Python. As the name suggests, we use them to store a collection of objects in a list (order matters)

We make a list using the [...] notation.

```
In [1]: boy_names = [
        "benny", "adam", "bobby",
        "randal", "timmy", "cartman",
        "morty", "junior-son",
        "voldemort", "boeta", "pula",
        "zane"
    ]
```

How long is this list?

```
In [2]: len(boy_names)
```

```
Out[2]: 12
```

Is "morty" in the list?

```
In [3]: "morty" in boy_names
```

```
Out[3]: True
```

Is "xavier" in the list?

```
In [4]: "xavier" in boy_names
```

```
Out[4]: False
```

We can access the zeroth element in the list.

```
In [5]: boy_names[0]
```

```
Out[5]: 'benny'
```

WARNING! Remember that we always start index from zero!

We will distinguish between the "first" and "oneth" element. "First element of boy_names" is ambiguous, do we mean boy_names[0] or boy_names[1] ?

By "oneth" or "1-th" element of boy_names we will always mean boy_names[1].

```
In [6]: boy_names[1]
```

```
Out[6]: 'adam'
```

We can access the last-th element in the list, by using the -1 index. (This is why we index starting from zero)

```
In [7]: boy_names[-1]
```

```
Out[7]: 'zane'
```

... and can access the 2nd last-th element with the -2 index

```
In [8]: boy_names[-2]
```

```
Out[8]: 'pula'
```

We can replace an element

```
In [9]: boy_names[1] = "adriaan"  
boy_names
```

```
Out[9]: ['benny',  
        'adriaan',  
        'bobby',  
        'randal',  
        'timmy',  
        'cartman',  
        'morty',  
        'junior-son',  
        'voldemort',  
        'boeta',  
        'pula',  
        'zane']
```

... and remove an element

```
In [10]: del boy_names[1]  
boy_names
```

```
Out[10]: ['benny',  
        'bobby',  
        'randal',  
        'timmy',  
        'cartman',  
        'morty',  
        'junior-son',  
        'voldemort',  
        'boeta',  
        'pula',  
        'zane']
```

... and append an element to the end of the list

```
In [11]: boy_names.append("sabelo")  
boy_names
```

```
Out[11]: ['benny',
          'bobby',
          'randal',
          'timmy',
          'cartman',
          'morty',
          'junior-son',
          'voldemort',
          'boeta',
          'pula',
          'zane',
          'sabelo']
```

3.1 Tuples

Tuples are like lists, but they are immutable. This means it is not possible to change tuples.

We make a tuples using the (...) notation

```
In [54]: boy_names_tuple = (
          "benny", "adam", "bobby",
          "randal", "timmy", "cartman",
          "morty", "junior-son",
          "voldemort", "boeta", "pula",
          "zane"
        )
```

How long is the tuple?

```
In [15]: len(boy_names_tuple)
```

```
Out[15]: 12
```

Is "morty" in the tuple?

```
In [16]: "morty" in boy_names_tuple
```

```
Out[16]: True
```

We can access the last-th element

```
In [24]: boy_names_tuple[-1]
```

```
Out[24]: 'zane'
```

... but we cannot change the tuple by replacing elements. Trying results in an error.

```
In [18]: # We cannot change a tuple, so the
          # following gives an error
          boy_names_tuple[1] = "adriaan"
```

```
-----
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-18-af5e5e9178b2> in <module>()
```

```

1 # We cannot change a tuple, so the
2 # following gives an error
----> 3 boy_names_tuple[1] = "adriaan"

```

TypeError: 'tuple' object does not support item assignment

3.2 List slicing

List slicing is an efficient method of cutting off parts of a list.

```

In [19]: girl_names = ["alice", "beatrice", "candy",
                      "dolly", "elaine", "francine", "geraldine"]

```

We use the ":" operator to make a slice. The following slice results in a new list containing the oneth, twoth, etc. elements:

```

In [22]: girl_names[1:]
Out[22]: ['beatrice', 'candy', 'dolly', 'elaine', 'francine', 'geraldine']

```

We can also slice from the other end. The following list contains everything up to the twoth element, and excludes the threeeth element onwards:

```

In [21]: girl_names[:3]
Out[21]: ['alice', 'beatrice', 'candy']

```

We can also slice using negative indices. The following list contains everything but the last-th element.

```

In [23]: girl_names[:-1]
Out[23]: ['alice', 'beatrice', 'candy', 'dolly', 'elaine', 'francine']

```

... and all but the second-last-th and last-th elements:

```

In [25]: girl_names[:-2]
Out[25]: ['alice', 'beatrice', 'candy', 'dolly', 'elaine']

```

The following list contains the oneth, twoth, threeeth, fourth, elements (excluding the fiveth element onward).

```

In [27]: girl_names[1:5]
Out[27]: ['beatrice', 'candy', 'dolly', 'elaine']

```

Slicing also works for tuples and for strings:

```

In [28]: name = "Marlon Brando"
          name[-4:]
Out[28]: 'ando'

```

3.3 Sorting

We very often want to sort lists. Python includes powerful methods to perform different kinds of sorting. We will work with the following list:

```
In [55]: boy_names = [  
    "benny", "adam", "bobby",  
    "randal", "timmy", "cartman",  
    "morty", "junior-son",  
    "voldemort", "boeta", "pula",  
    "zane"  
]
```

The default ordering for strings is alphabetically. We can just use the "sorted" function:

```
In [30]: sorted(boy_names)
```

```
Out[30]: ['adam',  
    'benny',  
    'bobby',  
    'boeta',  
    'cartman',  
    'junior-son',  
    'morty',  
    'pula',  
    'randal',  
    'timmy',  
    'voldemort',  
    'zane']
```

We can easily sort reverse-alphabetically

```
In [4]: sorted(boy_names, reverse=True)
```

```
Out[4]: ['zane',  
    'voldemort',  
    'timmy',  
    'randal',  
    'pula',  
    'morty',  
    'junior-son',  
    'cartman',  
    'boeta',  
    'bobby',  
    'benny',  
    'adam']
```

... or by length of the strings

```
In [5]: sorted(boy_names, key = len)
```

```
Out[5]: ['adam',  
    'pula',  
    'zane',  
    'benny',  
    'bobby',
```



```
'timmy',
'morty',
'boeta',
'randal',
'cartman',
'voldemort',
'junior-son']
```

We can sort with respect to any conceivable ordering. E.g. The following sorts the list alphabetically according to the one-th letter. (See the section on lambda expressions).

```
In [31]: sorted(boy_names, key = lambda item: item[1])
```

```
Out[31]: ['randal',
'cartman',
'zane',
'adam',
'benny',
'timmy',
'bobby',
'morty',
'voldemort',
'boeta',
'junior-son',
'pula']
```

3.4 Zipping

Zippping is an efficient way to combine two (or more) lists pairwise. Consider the two lists:

```
In [37]: girl_names = ["alice", "beatrice", "candy", "dolly", "elaine"]
their_ages = [10, 11, 10, 9, 8]
```

We can "zip" these two lists together to get a "zip" object (zip objects are iterable objects. Their purpose is for optimizing RAM usage).

```
In [38]: name_age_pairs = zip(girl_names, their_ages)
name_age_pairs
```

```
Out[38]: <zip at 0x7fed48073688>
```

The zip object can be converted to a list.

```
In [39]: list(name_age_pairs)
```

```
Out[39]: [('alice', 10), ('beatrice', 11), ('candy', 10), ('dolly', 9), ('elaine', 8)]
```

Warning! When zipping lists of unequal length the result will have the length of the shortest list:

```
In [40]: result = list(zip(["a","b","c"], [1,2,3,4,5,6]))
result
```

```
Out[40]: [('a', 1), ('b', 2), ('c', 3)]
```

We can also zip more than two lists:

```
In [41]: threezip = list(zip(["a","b","c"], [1,2,3], ["alpha", "beta", "gamma"]))
threezip
```

```
Out[41]: [('a', 1, 'alpha'), ('b', 2, 'beta'), ('c', 3, 'gamma')]
```

3.5 Unzipping

Unzipping is the opposite of zipping. I.e., Given a list of pairs, we can unzip the list into two lists: one list containing the first elements of the pairs and one list containing the second elements of the pairs.

Consider:

```
In [43]: name_age_pairs = [  
        ('alice', 10), ('beatrice', 11), ('candy', 10), #  
        ('dolly', 9), ('elaine', 8)  
        ]
```

... which we unzip (notice the "*"):

```
In [44]: unzipped_names, unzipped_ages = zip(*name_age_pairs)
```

Let's inspect the lists `unzipped_names` and `unzipped_ages`

```
In [47]: unzipped_names
```

```
Out[47]: ('alice', 'beatrice', 'candy', 'dolly', 'elaine')
```

```
In [48]: unzipped_ages
```

```
Out[48]: (10, 11, 10, 9, 8)
```

We can also unzip lists of triples:

```
In [49]: threezip = [('a', 1, 'alpha'), ('b', 2, 'beta'), ('c', 3, 'gamma')]  
abc, onetwothree, alphabeta gamma = zip(*threezip)
```

```
In [50]: abc
```

```
Out[50]: ('a', 'b', 'c')
```

```
In [51]: onetwothree
```

```
Out[51]: (1, 2, 3)
```

```
In [52]: alphabeta gamma
```

```
Out[52]: ('alpha', 'beta', 'gamma')
```

Chapter 4

Loops

Loops are used to perform a single operation over and over.

4.1 For-loops

The for-loop is the most used kind of loop. One can think of their operation as follows: "For every element in ...(container), do ...(action) on that element".

The "range" function is a useful container to loop over. The following example prints every number in the range 0,2,3,...,9:

```
In [1]: for i in range(10):  
        print(i)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

... do the same, but loop over 4,5,...,9

```
In [2]: for i in range(4, 10):  
        print(i)
```

```
4  
5  
6  
7  
8  
9
```

We are not limited to loop over "ranges", we can loop over any container. This is the preferred way to loop over a list in Python:

```
In [3]: girl_names = ["alice", "beatrice", "candy",  
                    "dolly", "elaine", "francine", "geraldine"]
```

```
    for name in girl_names:  
        print(name)
```

```
alice  
beatrice  
candy  
dolly  
elaine  
francine  
geraldine
```

We can loop in reverse order by just applying the "reversed" function to our list:

```
In [4]: for name in reversed(girl_names):  
        print(name)
```

```
geraldine  
francine  
elaine  
dolly  
candy  
beatrice  
alice
```

Often one want's to keep a running index. This is easily done with the "enumerate" function.

```
In [5]: for index, name in enumerate(girl_names):    #<-----(notice the unpacking)  
        print(index, " -> ", name)
```

```
0 ->  alice  
1 ->  beatrice  
2 ->  candy  
3 ->  dolly  
4 ->  elaine  
5 ->  francine  
6 ->  geraldine
```

We can also loop directly over zip objects using unpacking

```
In [14]: girl_names = ["alice", "beatrice", "candy",  
                    "dolly", "elaine", "francine", "geraldine"]  
        their_ages = [3,3,7,10,15,11,31]
```

```
    for name, age in zip(girl_names, their_ages):  
        print("name : ", name )  
        print("  age  : ", age)
```

```
name :  alice  
      age :  3  
name :  beatrice
```

```

    age : 3
name : candy
    age : 7
name : dolly
    age : 10
name : elaine
    age : 15
name : francine
    age : 11
name : geraldine
    age : 31

```

4.2 While loops

While loops are useful when we do not know before hand how many times a loop should execute. One can think of their operation as follows: "While ... (condition) is True, do ...(action)".

```

In [7]: number = 144
        while number % 2 == 0 :    # while number is divisible by 2, ...
            number = number // 2  # divide it by two

        print(number)

```

9

4.3 You should loop like a Pythonista not a C-snake.

Python is not like classic languages e.g., C. We should not use standard C-idioms in Python. Doing so will result in ugly, unreadable and un maintainable code.

DO NOT DO ANY OF THE FOLLOWING THINGS IN PYTHON. Compare the following bad looping idioms with the proper Pythonic looping idioms above.

Consider the lists

```

In [13]: girl_names = ["alice", "beatrice", "candy",
                       "dolly", "elaine", "francine", "geraldine"]
        their_ages = [3,3,7,10,15,11,31]

```

DO NOT Loop over a range object unnecessarily:

```

In [9]: for i in range(len(girl_names)):
        print(girl_names[i])

```

```

alice
beatrice
candy
dolly
elaine
francine
geraldine

```

DO NOT loop in reverse order by accessing indeces:

```
In [13]: for i in range(len(girl_names)):
         print(girl_names[len(girl_names) - i - 1])
```

```
geraldine
francine
elaine
dolly
candy
beatrice
alice
```

DO NOT keep a running index manually:

```
In [15]: index = 0
         for name in girl_names:
             print(index, " -> ", name)
             index = index + 1
```

```
0 ->  alice
1 ->  beatrice
2 ->  candy
3 ->  dolly
4 ->  elaine
5 ->  francine
6 ->  geraldine
```

DO NOT loop over two lists using indices:

```
In [12]: for i in range(min(len(girl_names), len(their_ages))):
         print("name : ", girl_names[i] )
         print("  age : ", their_ages[i])
```

```
name :  alice
      age :  3
name :  beatrice
      age :  3
name :  candy
      age :  7
name :  dolly
      age : 10
name :  elaine
      age : 15
name :  francine
      age : 11
name :  geraldine
      age : 31
```

Chapter 5

Computing with lists

5.1 sum, max, min

the sum, min and max functions allow for concisely expressing often occurring computations that we might perform on lists.

Consider the list:

```
In [3]: numbers = [3,5,10, 44,100,1,99]
```

...the sum of all the numbers in this list is

```
In [4]: sum(numbers)
```

```
Out[4]: 262
```

... the smallest is

```
In [5]: min(numbers)
```

```
Out[5]: 1
```

... and the largest is

```
In [6]: max(numbers)
```

```
Out[6]: 100
```

5.2 Computing with comprehensions: sum, max, min

We can perform computations with comprehensions. This is useful, because it makes our code easy to read and maintain.

We can compute the sum $1 + 2 + 3 + 4 + \dots + 100$:

```
In [9]: sum(i for i in range(1, 101)) # Why 101?
```

```
Out[9]: 5050
```

... or the sum of squares: $1 + 2^2 + 3^2 + 4^2, \dots, 100^2$:

```
In [6]: sum(i**2 for i in range(1, 101))
```

```
Out[6]: 338350
```

... or the sum of squares of even numbers: $2^2 + 4^2 + 6^2, \dots, 10000^2$:

```
In [8]: sum(i**2 for i in range(1, 10001) if i % 2 == 0)
```

```
Out[8]: 166716670000
```

We can also use the max or the min functions to find the largest or smallest elements in a list (or other container).

E.g., the smallest square number whose square lies in the interval [536,9000] is...

```
In [3]: min(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[3]: 24
```

... and the largest is

```
In [4]: max(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[4]: 94
```

If we define we can do more advanced searches as well.

E.g., finding the smallest prime number in the interval [536,9000] :

```
In [27]: def is_prime(n):  
         return not any (n % j==0 for j in range(2,n))
```

```
In [28]: min(p for p in range(536,9001) if is_prime(p))
```

```
Out[28]: 541
```

...and the largest

```
In [29]: max(p for p in range(536,9001) if is_prime(p))
```

```
Out[29]: 8999
```

5.3 More computing with min and max

We can use min and max with any objects that can be compared, like strings which are compared by their alphabetical order.

```
In [12]: names = ["randall", "jamie", "robert", "danaeris", "aegon", "tyrrion", "mother-of-dragons-breaker"]
```

The last name alphabetically is

```
In [14]: max(names)
```

```
Out[14]: 'tyrrion'
```

... the first

```
In [15]: min(names)
```

```
Out[15]: 'aegon'
```

... and the longest we can find by using len as a key function (see section on sorting)

```
In [30]: max(names, key=len)
```



```
Out[30]: 'mother-of-dragons-breaker-of-chains'
```

...and the shortest

```
In [31]: min(names, key=len)
```

```
Out[31]: 'jamie'
```

... and more advanced, the shortest and alphabetically first

```
In [32]: min(names, key=lambda x:(len(x), x))
```

```
Out[32]: 'aegon'
```

To understand this last example, understand that tuples are compared lexicographically.[\[https://en.wikipedia.org/wiki/Lexicographical_order\]](https://en.wikipedia.org/wiki/Lexicographical_order)

```
In [ ]:
```

Chapter 6

List comprehensions

List comprehensions is a concise way of constructing lists using a for-loop syntax.

Consider the list:

```
In [1]: girl_names = ["alice", "beatrice", "candy",  
                    "dolly", "elaine", "francine", "geraldine"]
```

We can use a list comprehension to make a new list containing the zeroth letter of each name in the list:

```
In [2]: first_letters = [name[0] for name in girl_names]  
first_letters
```

```
Out[2]: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

... or a list with the length of every name:

```
In [2]: length_of_names = [len(name) for name in girl_names]  
length_of_names
```

```
Out[2]: [5, 8, 5, 5, 6, 8, 9]
```

... or a list of name-length-pairs

```
In [3]: names_length_pairs = [ ( name, len(name) ) for name in girl_names]  
names_length_pairs
```

```
Out[3]: [('alice', 5),  
         ('beatrice', 8),  
         ('candy', 5),  
         ('dolly', 5),  
         ('elaine', 6),  
         ('francine', 8),  
         ('geraldine', 9)]
```

A useful feature is adding a conditional. The following makes a new list only containing the "long" names:

```
In [4]: only_long_names = [ name for name in girl_names if len(name) > 6 ]  
only_long_names
```

```
Out[4]: ['beatrice', 'francine', 'geraldine']
```

6.1 Computing with comprehensions: sum, min, max

We can perform computations with comprehensions. This is useful, because it makes our code easy to read and maintain.

We can compute the sum $1 + 2 + 3 + 4 + \dots + 100$:

```
In [9]: sum(i for i in range(1, 101)) # Why 101?
```

```
Out[9]: 5050
```

... or the sum of squares: $1 + 2^2 + 3^2 + 4^2, \dots, 100^2$:

```
In [6]: sum(i**2 for i in range(1, 101))
```

```
Out[6]: 338350
```

... or the sum of squares of even numbers: $2^2 + 4^2 + 6^2, \dots, 10000^2$:

```
In [8]: sum(i**2 for i in range(1, 10001) if i % 2 == 0)
```

```
Out[8]: 166716670000
```

We can also use the max or the min functions to find the largest or smallest elements in a list (or other container).

E.g., the smallest square number whose square lies in the interval [536,9000] is...

```
In [3]: min(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[3]: 24
```

... and the largest is

```
In [4]: max(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[4]: 94
```

If we define we can do more advanced searches as well.

E.g., finding the smallest prime number in the interval [536,9000] :

```
In [27]: def is_prime(n):  
         return not any (n % j==0 for j in range(2,n))
```

```
In [28]: min(p for p in range(536,9001) if is_prime(p))
```

```
Out[28]: 541
```

...and the largest

```
In [29]: max(p for p in range(536,9001) if is_prime(p))
```

```
Out[29]: 8999
```

6.2 More computing with min and max

We can use min and max with any objects that can be compared, like strings which are compared by their alphabetical order.

```
In [12]: names = ["randall", "jamie", "robert", "danaeris", "aegon", "tyrrion", "mother-of-dragons-breaker-of-chains"]
```

The last name alphabetically is

```
In [14]: max(names)
```

```
Out[14]: 'tyrrion'
```

... the first

```
In [15]: min(names)
```

```
Out[15]: 'aegon'
```

... and the longest we can find by using len as a key function (see section on sorting)

```
In [30]: max(names, key=len)
```

```
Out[30]: 'mother-of-dragons-breaker-of-chains'
```

...and the shortest

```
In [31]: min(names, key=len)
```

```
Out[31]: 'jamie'
```

... and more advanced, the shortest and alphabetically first

```
In [32]: min(names, key=lambda x:(len(x), x))
```

```
Out[32]: 'aegon'
```

To understand this last example, understand that Python tuples are compared lexicographically. See: https://en.wikipedia.org/wiki/Lexicographical_order

```
In [ ]:
```

Chapter 7

Functions

Functions allow for the easy reuse of bits of code. They take parameters/input, and can return a result. Functions are defined using the *def* keyword.

7.1 Example: Say hello

We define a function that takes *name* as parameter, and prints a greeting for that name:

```
In [4]: def say_hello_to(name):  
        print("Hello " + name + "!")
```

We can now call this function with different inputs:

```
In [6]: say_hello_to("World")  
        say_hello_to("Gary")  
        say_hello_to("Alice")  
        say_hello_to("Crocubot")
```

```
Hello World!  
Hello Gary!  
Hello Alice!  
Hello Crocubot!
```

7.2 Example: Divisible by 11

We define a function that takes *number* as input and *returns* whether or not the number is divisible by 11

```
In [7]: def is_divisible_by_11(number):  
        return number % 11 == 0
```

Lets check which numbers of 10,11,12,...,24 are divisible by 11

```
In [8]: for number in range(10, 25):  
        print(number, "is divisible by 11 : ", is_divisible_by_11(number))
```

```
10 is divisible by 11 : False  
11 is divisible by 11 : True  
12 is divisible by 11 : False  
13 is divisible by 11 : False  
14 is divisible by 11 : False
```

```
15 is divisible by 11 : False
16 is divisible by 11 : False
17 is divisible by 11 : False
18 is divisible by 11 : False
19 is divisible by 11 : False
20 is divisible by 11 : False
21 is divisible by 11 : False
22 is divisible by 11 : True
23 is divisible by 11 : False
24 is divisible by 11 : False
```

7.3 Example: The Collatz function

Example: We define the *collatz* function according to the following specification.

Input:

- A number n .

Output:

- return 1 if $n = 1$
- return $n/2$ if n is even
- return $3n + 1$ if n is odd

```
In [9]: def collatz(number):
        if number == 1:
            return number
        elif number % 2 == 0:
            return number // 2
        else:
            return 3*number + 1
```

Let's try it out on 3,11,24 and 65

```
In [10]: for n in [3, 11, 24, 65]:
          print(n, " -> ",collatz(n))
```

```
3  ->  10
11  ->  34
24  ->  12
65  -> 196
```

Let's repeatedly apply the collatz function to a number using a while loop. We always tend to get back to 1... why is that?

See https://en.wikipedia.org/wiki/Collatz_conjecture

```
In [12]: current_number = 15
        while current_number != 1:
            current_number = collatz(current_number)
            print(current_number)
```

46
23
70
35
106
53
160
80
40
20
10
5
16
8
4
2
1

7.4 Recursion

Recursion is what happens when a function calls itself.

7.4.1 Example: Fibonacci numbers

A good example of recursion is the process of generating Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... These are formally defined as the sequence (f_n) with $f_1 := 1$, $f_2 := 1$ and $f_n := f_{n-1} + f_{n-2}$ for all $n \in \{3, 4, 5, \dots\}$

Pay attention how the following function calls itself:

```
In [21]: def fibonacci(n):  
        if n == 1:  
            return 1  
        elif n == 2:  
            return 1  
        else:  
            return fibonacci(n-1) + fibonacci(n-2)
```

Let's compute the first 20 fibonacci numbers

```
In [14]: for i in range(1,20):  
        print(fibonacci(i))
```

1
1
2
3
5
8
13
21
34

```
55
89
144
233
377
610
987
1597
2584
4181
```

7.5 Making functions with lambda expressions

Very simple functions can be defined using *lambda* expressions. We've already briefly encountered lambda expressions in the section on sorting.

```
In [16]: f = lambda x: 3*x
```

Make sure you understand why $f(4)=12$

```
In [19]: f(4)
```

```
Out[19]: 12
```

...and $f('a')$ ='aaa'

```
In [20]: f("a")
```

```
Out[20]: 'aaa'
```

7.6 docstrings

One's code is usually used by other people. These people might need to know what a function you wrote does. One may do this by writing a short explanation in a *docstring* in the first line of the function definition. This can be accessed by calling the help function on an object.

```
In [25]: def fibonacci_undocumented(n):
        if n == 1:
            return 1
        elif n == 2:
            return 1
        else:
            return fibonacci(n-1) + fibonacci(n-2)
```

No help is forthcoming...

```
In [26]: help(fibonacci_undocumented)
```

Help on function fibonacci_undocumented in module __main__:

```
fibonacci_undocumented(n)
```


... unless we give it:

```
In [29]: def fibonacci(n):  
        """  
        Returns the nth fibonacci number.  
  
        Input: n  
        Output: the nth fibonacci number  
  
        E.g. fibonacci(1) = 1  
             fibonacci(2) = 1  
             fibonacci(3) = 2  
        ...  
        """  
        if n == 1:  
            return 1  
        elif n == 2:  
            return 1  
        else:  
            return fibonacci(n-1) + fibonacci(n-2)
```

```
In [30]: help(fibonacci)
```

Help on function fibonacci in module __main__:

```
fibonacci(n)  
    Returns the nth fibonacci number.  
  
    Input: n  
    Output: the nth fibonacci number  
  
    E.g. fibonacci(1) = 1  
         fibonacci(2) = 1  
         fibonacci(3) = 2  
    ...
```

Chapter 8

The 'any' and 'all' functions

Sometimes one is required to decide if a number of statements in a list are *all* true.

```
In [4]: all([True, True, True, True]) # All true? Yes.
```

```
Out[4]: True
```

```
In [7]: all([True, True, False, True]) # All true? No.
```

```
Out[7]: False
```

... and sometimes one is required to decide if *at least one* from a number of statements is true:

```
In [15]: any([True, False, False, True]) # Is at least one statement true? Yes.
```

```
Out[15]: True
```

```
In [10]: any([False, False, False, False]) # Is at least one statement true? No.
```

```
Out[10]: False
```

Examples

The following examples illustrate how the *all* and *any* functions can be used.

```
In [2]: # Do all the letters "a", "b", "l" occur the phrase "mary had a little lamb"?  
# Yes. So the following evaluates to True.  
all([letter in "mary had a little lamb" for letter in ["a", "b", "l"]])
```

```
Out[2]: True
```

```
In [4]: # Do all the letters "a", "b", "q" occur the phrase "the quick brown fox"?  
# No. The letter "a" does not occur, so the following evaluates to False.  
all([letter in "the quick brown fox" for letter in ["a", "b", "q"]])
```

```
Out[4]: False
```

```
In [7]: # Does at least one of the letters "a", "b", "z" occur the phrase "the quick brown fox"?  
# Yes. The letter "b" occurs, so the following evaluates to True.  
any([letter in "the quick brown fox" for letter in ["a", "b", "z"]])
```

```
Out[7]: True
```

```
In [8]: # Does at least one of the letters "z", "q", "p" occur the phrase "mary had a little lamb"?  
# No. none of the letters occur, so the following evaluates to False.  
any([letter in "mary had a little lamb" for letter in ["z", "p", "q"]])
```

```
Out[8]: False
```

Chapter 9

Computing with lists

9.1 sum, max, min

the sum, min and max functions allow for concisely expressing often occurring computations that we might perform on lists.

Consider the list:

```
In [3]: numbers = [3,5,10, 44,100,1,99]
```

...the sum of all the numbers in this list is

```
In [4]: sum(numbers)
```

```
Out[4]: 262
```

... the smallest is

```
In [5]: min(numbers)
```

```
Out[5]: 1
```

... and the largest is

```
In [6]: max(numbers)
```

```
Out[6]: 100
```

9.2 Computing with comprehensions: sum, min, max

We can perform computations with comprehensions. This is useful, because it makes our code easy to read and maintain.

We can compute the sum $1 + 2 + 3 + 4 + \dots + 100$:

```
In [9]: sum(i for i in range(1, 101)) # Why 101?
```

```
Out[9]: 5050
```

... or the sum of squares: $1 + 2^2 + 3^2 + 4^2, \dots, 100^2$:

```
In [6]: sum(i**2 for i in range(1, 101))
```

```
Out[6]: 338350
```

... or the sum of squares of even numbers: $2^2 + 4^2 + 6^2, \dots, 10000^2$:

```
In [8]: sum(i**2 for i in range(1, 10001) if i % 2 == 0)
```

```
Out[8]: 166716670000
```

We can also use the max or the min functions to find the largest or smallest elements in a list (or other container).

E.g., the smallest square number whose square lies in the interval [536,9000] is...

```
In [3]: min(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[3]: 24
```

... and the largest is

```
In [4]: max(i for i in range(1,9001) if 536 <= i**2 <= 9000 )
```

```
Out[4]: 94
```

If we define we can do more advanced searches as well.

E.g., finding the smallest prime number in the interval [536,9000] :

```
In [27]: def is_prime(n):  
         return not any (n % j==0 for j in range(2,n))
```

```
In [28]: min(p for p in range(536,9001) if is_prime(p))
```

```
Out[28]: 541
```

...and the largest

```
In [29]: max(p for p in range(536,9001) if is_prime(p))
```

```
Out[29]: 8999
```

9.3 More computing with min and max

We can use min and max with any objects that can be compared, like strings which are compared by their alphabetical order.

```
In [12]: names = ["randall", "jamie", "robert", "danaeris", "aegon", "tyrrion", "mother-of-dragons-breaker"]
```

The last name alphabetically is

```
In [14]: max(names)
```

```
Out[14]: 'tyrrion'
```

... the first

```
In [15]: min(names)
```

```
Out[15]: 'aegon'
```

... and the longest we can find by using len as a key function (see section on sorting)

```
In [30]: max(names, key=len)
```

```
Out[30]: 'mother-of-dragons-breaker-of-chains'
```

...and the shortest

```
In [31]: min(names, key=len)
```

```
Out[31]: 'jamie'
```

... and more advanced, the shortest and alphabetically first

```
In [32]: min(names, key=lambda x:(len(x), x))
```

```
Out[32]: 'aegon'
```

To understand this last example, understand that tuples are compared lexicographically.[\[https://en.wikipedia.org/wiki/Lexicographical_order\]](https://en.wikipedia.org/wiki/Lexicographical_order)

```
In [ ]:
```

Chapter 10

Dictionaries

Dictionaries are datastructures that map one object to another. We create a dictionary using the `{ ... : ... }` notation.

Consider the dictionary:

```
In [1]: surname_dictionary = {
        # key    : #value,
        "kevin" : "de koker",
        "john"  : "mphako",
        "alice" : "munro",
        "doris" : "lessing",
    }
    surname_dictionary
```

```
Out[1]: {'alice': 'munro', 'doris': 'lessing', 'john': 'mphako', 'kevin': 'de koker'}
```

We can access a *value* associated to a specific *key*:

```
In [2]: surname_dictionary["kevin"]
```

```
Out[2]: 'de koker'
```

```
In [3]: surname_dictionary["alice"]
```

```
Out[3]: 'munro'
```

An error is raised if the key is not in the dictionary:

```
In [4]: surname_dictionary["bobby"]
```

```
-----
KeyError                                Traceback (most recent call last)

<ipython-input-4-aebb82606656> in <module>()
----> 1 surname_dictionary["bobby"]

KeyError: 'bobby'
```

We can ask if a key is in the dictionary:

```
In [5]: "kevin" in surname_dictionary
```

```
Out[5]: True
```

```
In [6]: "bobby" in surname_dictionary
```

```
Out[6]: False
```

... the *in* operator only checks keys, not values:

```
In [7]: "de koker" in surname_dictionary
```

```
Out[7]: False
```

We can add elements:

```
In [8]: surname_dictionary["katie"] = "van der merwe"
surname_dictionary
```

```
Out[8]: {'alice': 'munro',
        'doris': 'lessing',
        'john': 'mphako',
        'katie': 'van der merwe',
        'kevin': 'de koker'}
```

... and remove elements:

```
In [10]: del surname_dictionary["alice"]
surname_dictionary
```

```
Out[10]: {'doris': 'lessing',
        'john': 'mphako',
        'katie': 'van der merwe',
        'kevin': 'de koker'}
```

Iterating over a dictionary, iterates over the keys:

```
In [11]: for key in surname_dictionary:
        print(key)
```

```
doris
john
kevin
katie
```

... but we can also iterate over the values using *.values()*:

```
In [12]: for key in surname_dictionary.values():
        print(key)
```

```
lessing
mphako
de koker
van der merwe
```

... or we can iterate over key-value pairs using *.items()*:

```
In [13]: for pair in surname_dictionary.items():  
         print(pair)
```

```
('doris', 'lessing')  
('john', 'mphako')  
('kevin', 'de koker')  
('katie', 'van der merwe')
```

It is often useful to unpack such pairs:

```
In [14]: for firstname, lastname in surname_dictionary.items():  
         print(firstname, "-->", lastname[0])
```

```
doris --> l  
john --> m  
kevin --> d  
katie --> v
```


Chapter 11

Dictionary comprehensions

Dictionary comprehension is a concise way to construct dictionaries using a for-loop syntax.
Consider:

```
In [1]: surname_dictionary = {  
        # key    : #value,  
        "kevin" : "de koker",  
        "john"  : "mphako",  
        "alice" : "munro",  
        "doris" : "lessing",  
    }  
    surname_dictionary
```

```
Out[1]: {'alice': 'munro', 'doris': 'lessing', 'john': 'mphako', 'kevin': 'de koker'}
```

We construct a dictionary which maps a name to the length of the surname.

```
In [2]: length_of_surname_dictionary = {  
        firstname : len(lastname) for firstname, lastname in surname_dictionary.items()  
    }  
    length_of_surname_dictionary
```

```
Out[2]: {'alice': 5, 'doris': 7, 'john': 6, 'kevin': 8}
```

We construct a dictionary which filtered all items whose last name start with "m"

```
In [3]: last_name_starts_with_m = {  
        firstname : lastname  
        for firstname, lastname in surname_dictionary.items()  
        if "m" == lastname[0]  
    }  
    last_name_starts_with_m
```

```
Out[3]: {'alice': 'munro', 'john': 'mphako'}
```

Chapter 12

Importing modules and interactive help

Not all Python functionality is builtin. Extra functionality is provided in *modules*. To use the extra functionality provided by a module we must *import* the module.

The syntax for importing modules are:

```
import ...  
from ... import ...  
import ... as ...
```

Let's import the *math* module:

```
In [2]: import math
```

We can see what objects the *math* module provides by calling the *dir* method on it:

```
In [3]: dir(math)
```

```
Out[3]: ['__doc__',  
         '__loader__',  
         '__name__',  
         '__package__',  
         '__spec__',  
         'acos',  
         'acosh',  
         'asin',  
         'asinh',  
         'atan',  
         'atan2',  
         'atanh',  
         'ceil',  
         'copysign',  
         'cos',  
         'cosh',  
         'degrees',  
         'e',  
         'erf',  
         'erfc',  
         'exp',  
         'expm1',  
         'fabs',  
         'factorial',  
         'floor',
```

```
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'hypot',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'modf',
'nan',
'pi',
'pow',
'radians',
'sin',
'sinh',
'sqrt',
'tan',
'tanh',
'trunc']
```

If we need to know more about an object, then we can the *help* function on it:

```
In [4]: help(math.acos)
```

Help on built-in function acos in module math:

```
acos(...)
    acos(x)
```

Return the arc cosine (measured in radians) of x.

```
In [6]: help(math.radians)
```

Help on built-in function radians in module math:

```
radians(...)
    radians(x)
```

Convert angle x from degrees to radians.

With the math module imported, we can access its contents and call the functions it defines:

```
In [7]: math.pi
```

```
Out[7]: 3.141592653589793
```

```
In [8]: math.acos(-1)
```

```
Out[8]: 3.141592653589793
```

```
In [9]: math.sin(math.radians(90))
```

```
Out[9]: 1.0
```

Chapter 13

Sympy

Sympy is an external Python module that allows for symbolic computations like solving equations, differentiation and integration.

We import the *sympy* module

```
In [2]: import sympy
```

If we want to have pretty output inside a Jupyter notebook, we call `sympy.init_printing`

```
In [3]: sympy.init_printing()
```

We can define symbols using the `sympy.symbols` function:

```
In [4]: x,y = sympy.symbols("x y")
```

... or we can import standard symbols from the `sympy.abc` module

```
In [5]: from sympy.abc import x,y
```

With these symbols, we can define an algebraic expression in the variables `x` and `y`

```
In [6]: an_expression = sympy.sin(x**2 - x - 1 + sympy.acos(y))
        an_expression
```

```
Out[6]:
```

$$\sin\left(x^2 - x + \arccos(y) - 1\right)$$

13.1 Expanding, factoring and simplifying expressions

We can expand expressions using `sympy.expand`

```
In [7]: sympy.expand( (x+4)*(x-6) )
```

```
Out[7]:
```

$$x^2 - 2x - 24$$

We can factor expressions using `sympy.factor`

```
In [8]: sympy.factor( x**2-x-20 )
```

Out [8]:

$$(x - 5)(x + 4)$$

We can make more complicated expressions ...

```
In [47]: (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
```

Out [47]:

$$\frac{x^3 + x^2 - x - 1}{x^2 + 2x + 1}$$

... and simplify them using `sympy.simplify`

```
In [9]: sympy.simplify( (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1) )
```

Out [9]:

$$x - 1$$

13.2 Substituting values into expressions

Let's define the quadratic expression $x^2 - x - 1$

```
In [10]: from sympy.abc import x
         quadratic_expression = x**2 - x - 1
```

... and substitute the value 1 for the symbol x using the `.subs` function.
Notice the dictionary! Make sure you understand why the result is -1.

```
In [12]: quadratic_expression.subs({x : 1})
```

Out [12]:

$$-1$$

We substitute the value -2 for the symbol x using the `.subs` function. Make sure you understand why the result is 5.

```
In [15]: quadratic_expression.subs({x : -2})
```

Out [15]:

$$5$$

13.3 Solving equations

We can solve equations with `sympy`.

WARNING: We cannot use `"=`" or `"=="` to define equations, we must use `sympy.Eq`.

We make the equation $x^2 - x - 1 = 0$.

```
In [17]: import sympy
         from sympy.abc import x

         sympy.Eq(x**2 - x - 1, 0)
```

Out [17]:

$$x^2 - x - 1 = 0$$

... and solve for x in this equation by calling the `sympy.solve` function

```
In [18]: sympy.solve(sympy.Eq(x**2 - x - 1, 0), x)
```

Out [18]:

$$\left[\frac{1}{2} + \frac{\sqrt{5}}{2}, -\frac{\sqrt{5}}{2} + \frac{1}{2} \right]$$

... by just providing an expression to `sympy.solve`, it solves the equation `expression=0`.

```
In [19]: sympy.solve(x**2 - x - 1, x)
```

Out [19]:

$$\left[\frac{1}{2} + \frac{\sqrt{5}}{2}, -\frac{\sqrt{5}}{2} + \frac{1}{2} \right]$$

```
In [ ]:
```

We can solve more complicated equations. Let's solve for θ in:

$$\cos(\theta) = \sin(\theta)$$

The equation has infinitely many solutions in θ . However, `sympy.solve` only gives two:

```
In [24]: from sympy.abc import theta
         sympy.solve(sympy.Eq(sympy.cos(theta), sympy.sin(theta)), theta)
```

Out [24]:

$$\left[-\frac{3\pi}{4}, \frac{\pi}{4} \right]$$

... `sympy.solveset` gives all infinitely many solutions

```
In [25]: sympy.solveset(sympy.Eq(sympy.cos(theta), sympy.sin(theta)), theta)
```

Out [25]:

$$\left\{ 2n\pi + \frac{5\pi}{4} \mid n \in \mathbb{Z} \right\} \cup \left\{ 2n\pi + \frac{\pi}{4} \mid n \in \mathbb{Z} \right\}$$

13.4 Solving systems of equations

We can also solve systems of equations like the following in x and y :

$$2x + 3y = 1$$

$$3x + 2y = 2$$

```
In [26]: from sympy.abc import x,y
         import sympy

         sympy.solve([
             sympy.Eq(2*x + 3*y, 1),
             sympy.Eq(3*x + 2*y, 2)
         ], [x,y])
```

Out [26]:

$$\left\{ x : \frac{4}{5}, \quad y : -\frac{1}{5} \right\}$$

13.5 Generating complicated expressions

We can use functions to generate complicated expressions.

Let's define a function P that, for any number n , returns a polynomial of the form

$$\sum_{k=0}^n kx^k$$

```
In [59]: import sympy
         from sympy.abc import x

         def P(n):
             return sum( k * x**k for k in range(n+1) )
```

We can now obtain the 6th degree polynomial of the given form by calling $P(6)$

```
In [60]: P(6)
```

Out [60]:

$$6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x$$

... or the 10th degree polynomial by calling $P(10)$

```
In [33]: P(10)
```

Out [33]:

$$10x^{10} + 9x^9 + 8x^8 + 7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x$$

For fun, let's solve the equation $4x^4 + 3x^3 + 2x^2 + x = 0$.

```
In [35]: sympy.solve( P(4), x)
```

Out [35]:

$$\left[0, -\frac{1}{4} + \frac{5}{16 \left(-\frac{1}{2} - \frac{\sqrt{3}i}{2} \right) \sqrt[3]{\frac{135}{64} + \frac{15\sqrt{6}}{16}}} - \frac{1}{3} \left(-\frac{1}{2} - \frac{\sqrt{3}i}{2} \right) \sqrt[3]{\frac{135}{64} + \frac{15\sqrt{6}}{16}}, -\frac{1}{4} - \frac{1}{3} \left(-\frac{1}{2} + \frac{\sqrt{3}i}{2} \right) \sqrt[3]{\frac{135}{64} + \frac{15\sqrt{6}}{16}} + \frac{1}{16} \right]$$

13.6 Numerical approximation

Sometimes we want numerical approximations to mathematical constants. We can compute them to arbitrary precision with the sympy's `.n` function.

```
In [36]: import sympy
```

```
In [37]: sympy.pi
```


Out [37]:

$$\pi$$

... pi approximated to 50 digits is:

In [40]: `sympy.pi.n(50)`

Out [40]:

3.1415926535897932384626433832795028841971693993751

In [41]: `sympy.sqrt(2)`

Out [41]:

$$\sqrt{2}$$

$\sqrt{2}$ two approximated to 50 digits is

In [42]: `sympy.sqrt(2).n(50)`

Out [42]:

1.4142135623730950488016887242096980785696718753769

... we can even find approximations to more complicated expressions:

In [44]: `sympy.exp(sympy.root(sympy.sqrt(2)-1,5))`

Out [44]:

$$e^{\sqrt[5]{-1+\sqrt{2}}}$$

In [45]: `sympy.exp(sympy.root(sympy.sqrt(2)-1,5)).n(50)`

Out [45]:

2.3126351501944463406364037678832846493032008107479

13.7 Symbolic differentiation

We can use sympy to compute derivatives of expressions using the `sympy.diff` function.

Let's compute $\frac{d}{dx} (x^4 + x^3 + x^2 + x + 1)$

In [46]: `import sympy
from sympy.abc import x`

`sympy.diff(x**4+x**3+x**2+x+1, x)`

Out [46]:

$$4x^3 + 3x^2 + 2x + 1$$

... or the more complicated derivative $\frac{d}{dx} \left((x^4 + x^3 + x^2 + x + 1)e^{x^2 + \sin(x^2)} \right)$

In [47]: `sympy.diff((x**4+x**3+x**2+x+1)*sympy.exp(x**2 + sympy.sin(x**2)), x)`

Out [47]:

$$(2x \cos(x^2) + 2x) (x^4 + x^3 + x^2 + x + 1) e^{x^2 + \sin(x^2)} + (4x^3 + 3x^2 + 2x + 1) e^{x^2 + \sin(x^2)}$$

13.8 Symbolic integration

We can use sympy to compute derivatives of expressions using the `sympy.integrate` function.

Let's compute $\int (x^2 - x - 1) dx$:

```
In [48]: import sympy
         from sympy.abc import x

         sympy.integrate(x**2 - x - 1, x)
```

Out[48]:

$$\frac{x^3}{3} - \frac{x^2}{2} - x$$

```
In [28]: # ... another example, that illustrates integration by parts:
```

... or another example (notice the integration by parts): $\int x e^x dx$

```
In [49]: sympy.integrate(x * sympy.exp(x), x)
```

Out[49]:

$$(x - 1) e^x$$

We can also compute definite integrals. E.g., $\int_0^5 x e^x dx$.

```
In [51]: sympy.integrate(x * sympy.exp(x), (x, 0, 5))
```

Out[51]:

$$1 + 4e^5$$

13.9 Making functions out of expressions

Especially for plotting, it is useful to be able to make a function out of a sympy expression. We can do this with the `sympy.lambdify` function

```
In [52]: import sympy
         from sympy.abc import x,y

         cubic = x**3 - x**2 + x + 3
         f = sympy.lambdify([x], cubic)
```

```
In [ ]:
```

Now we have the function $f(x) := x^3 - x^2 + x + 3$ and we can call it:

```
In [56]: f(2)
```

Out[56]:

$$9$$

```
In [57]: f(x)
```

Out [57]:

$$x^3 - x^2 + x + 3$$

In [58]: f(y)

Out [58]:

$$y^3 - y^2 + y + 3$$

Chapter 14

Numpy

Numpy is a widely used external package for doing matrix computations. It is designed to be very fast.

14.1 Arrays

The array is the basic datastructure of numpy. We can think of them as vectors.

```
In [3]: import numpy as np
```

```
a = np.array([1,2,3])
b = np.array([4,3,3])
a + b
```

```
Out[3]: array([5, 5, 6])
```

... we can compute the dot product:

```
In [4]: a.dot(b) # 4 + 6 + 9
```

```
Out[4]: 19
```

14.2 Numpy functions

Numpy provides many mathematical functions like `numpy.sin`, `numpy.cos`, etc. When applying these to arrays, they are applied entry-wise. This is useful for plotting.

```
In [5]: import numpy as np
```

```
a = np.array([0,1,2,3,4,5,6,7,8,9,10])
b = np.sin(a)
c = np.sqrt(a)
```

```
In [6]: # b == [sin(0), sin(1), sin(2), ... , sin(10)]
b
```

```
Out[6]: array([ 0.          ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
        -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849,
        -0.54402111])
```

```
In [7]: # c == [sqrt(0), sqrt(1), sqrt(2), ... , sqrt(10)]
c
```

```
Out[7]: array([ 0.          ,  1.          ,  1.41421356,  1.73205081,  2.          ,
                2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.          ,
                3.16227766])
```

14.3 Matrices

Matrices can be represented as 2D numpy arrays

```
In [12]: import numpy as np
```

```
M = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
])
```

We can access columns, e.g., the twoth column:

```
In [13]: M[:,2]
```

```
Out[13]: array([3, 6, 9])
```

We can access columns, e.g., the oneth row:

```
In [15]: M[1,:]
```

```
Out[15]: array([4, 5, 6])
```

We can compute multiply a matrix with a vector using the `.dot` function:

```
In [17]: v = np.array([1,2,3,])
         M.dot(v)
```

```
Out[17]: array([14, 32, 50])
```

We can multiply a matrix with another matrix, also using the `.dot` function :

```
In [18]: M.dot(M)
```

```
Out[18]: array([[ 30,  36,  42],
                [ 66,  81,  96],
                [102, 126, 150]])
```

Waring: The `*` operator does entrywise multiplication!

```
In [19]: M*M
```

```
Out[19]: array([[ 1,  4,  9],
                [16, 25, 36],
                [49, 64, 81]])
```

14.4 Matrix row/column operations

Consider:

```
In [24]: import numpy as np
```

```
M = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
])
```

We can easily perform row/column swaps.

The following swaps the zeroth and oneth rows of M:

```
In [25]: M[[1,0],:] = M[[0,1],:]
```

```
In [26]: M
```

```
Out[26]: array([[4, 5, 6],
               [1, 2, 3],
               [7, 8, 9]])
```

The following swaps the oneth and twoth columns of M:

```
In [27]: M[:,[1,2]] = M[:,[2,1]]
```

```
In [29]: M
```

```
Out[29]: array([[4, 6, 5],
               [1, 3, 2],
               [7, 9, 8]])
```

... and we can perform elementary row operations (e.g., for implementing Gauss elimination).

The following replaces the twoth row with 4 times the twoth row - 7 times the zeroth row

```
In [30]: M[2,:] = 4*M[2,:] - 7*M[0,:]
```

```
In [31]: M
```

```
Out[31]: array([[ 4,  6,  5],
               [ 1,  3,  2],
               [ 0, -6, -3]])
```

Transposing is easy.

```
In [33]: M.transpose()
```

```
Out[33]: array([[ 4,  1,  0],
               [ 6,  3, -6],
               [ 5,  2, -3]])
```